Homework 6: Scheduling

Course: CS 221 Spring 2019
Name: Bryan Yaggi

What courses should you take in a given quarter? Answering this question requires balancing your interests, satisfying prerequisite chains, graduation requirements, availability of courses; this can be a complex tedious process. In this assignment, you will write a program that does automatic course scheduling for you based on your preferences and constraints. The program will cast the course scheduling problem (CSP) as a constraint satisfaction problem (CSP) and then use backtracking search to solve that CSP to give you your optimal course schedule.

You will first get yourself familiar with the basics of CSPs in Problem 0. In Problem 1, you will implement two of the three heuristics you learned from the lectures that will make CSP solving much faster. In Problem 2, you will add a helper function to reduce n-ary factors to unary and binary factors. Lastly, in Problem 3, you will create the course scheduling CSP and solve it using the code from previous parts.

Problem 0: CSP Basics

(a) Let's create a CSP. Suppose you have n light bulbs, where each light bulb $i = 1, \ldots, n$ is initially off. You also have m buttons which control the lights. For each button $j = 1, \ldots, m$, we know the subset $T_j \subseteq \{1, \ldots, n\}$ of light bulbs that it controls. When button j is pressed, it toggles the state of each light bulb in T_j (For example, if $3 \in T_j$ and light bulb 3 is off, then after the button is pressed, light bulb 3 will be on, and vice versa).

Your goal is to turn on all the light bulbs by pressing a subset of the buttons. Construct a CSP to solve this problem. Your CSP should have m variables and n constraints. For this problem only, you can use n-ary constraints. Describe your CSP precisely and concisely. You need to specify the variables with their domain, and the constraints with their scope and expression. Make sure to include T_j in your answer.

There are m variables, one for each button. The domain consists of 1 or 0, indicating whether the button is pressed or not.

$$X = (X_1, \ldots, X_m), where\ X_j \in \{0, 1\}$$

There are n factors, one for each light. The factor will be 1 if the number of pressed buttons controlling the light is odd.

$$f_i = \mathbb{1}\left[\left(\sum_{j: i \in T_j}^{m} X_j\right) \mod 2\right]$$

(b) Let's consider a simple CSP with 3 variables and 2 binary factors:

$$X_1 \longrightarrow \boxed{t_1} \longrightarrow X_2 \longrightarrow \boxed{t_2} \longrightarrow X_3$$

where $X_1, X_2, X_3 \in \{0, 1\}$ and t_1, t_2 are XOR functions (that is $t_1(X) = X_1 \oplus X_2$ and $t_2(X) = X_2 \oplus X_3$).

i. How many consistent assignments are there for this CSP?

X_1	X_2	X_3	t_1	t_2	*weight*
0	0	0	0	0	0
1	0	0	1	0	0
0	1	0	1	1	1
1	1	0	0	1	0
0	0	1	0	1	0
1	0	1	1	1	1
0	1	1	1	0	0
1	1	1	0	0	0

There are 2 consistent assignments: $x = \{X_1 = 0, X_2 = 1, X_3 = 0\}$ and $x = \{X_1 = 1, X_2 = 0, X_3 = 1\}$.

ii. To see why variable ordering is important, let's use backtracking search to solve the CSP without using any heuristics (MCV, LCV, AC-3) or lookahead. How many times will `backtrack()` be called to get all consistent assignments if we use the fixed ordering X_1, X_3, X_2? Draw the call stack for `backtrack()`. (You should use the Backtrack algorithm from the slides. The initial arguments are $x = \emptyset, w = 1$, and the original Domain.)

In the code, this number will be stored in `BacktrackingSearch.numOperations`.

Use domain ordering $Domain = \{0, 1\}$. The top of the stack is on the bottom.

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0, X_3 = 0\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0, X_3 = 0, X_2 = 1\}$`, `$w = 1$`)` *consistent assignment found*

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0, X_3 = 1\}$`, `$w = 1$`)`

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1, X_3 = 0\}$`, `$w = 1$`)`

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1, X_3 = 1\}$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1, X_3 = 1, X_2 = 0\}$`, `$w = 1$`)` *consistent assignment found*

`backtrack()` is called 9 times.

iii. To see why lookahead can be useful, let's do it again with the ordering X_1, X_3, X_2 and AC-3. How many times will Backtrack be called to get all consistent assignments? Draw the call stack for `backtrack()`.

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 0\}$`, `$w = 1$`)` $Domain_3 = \{0\}$ $Domain_2 = \{1\}$
`backtrack(`$x = \{X_1 = 0, X_3 = 0\}$`, `$w = 1$`)` $Domain_2 = \{1\}$
`backtrack(`$x = \{X_1 = 0, X_3 = 0, X_2 = 1\}$`, `$w = 1$`)` *consistent assignment found*

`backtrack(`$x = \emptyset$`, `$w = 1$`)`
`backtrack(`$x = \{X_1 = 1\}$`, `$w = 1$`)` $Domain_3 = \{1\}$ $Domain_2 = \{0\}$
`backtrack(`$x = \{X_1 = 1, X_3 = 1\}$`, `$w = 1$`)` $Domain_2 = \{0\}$
`backtrack(`$x = \{X_1 = 1, X_3 = 1, X_2 = 0\}$`, `$w = 1$`)` *consistent assignment found*

`backtrack()` is called 7 times.

(c) coding